The Dynamic Composition of Abductive Agents in *ALIAS*

A. Ciampolini, E. Lamma, P. Mello and P. Torroni

DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
{aciampolini,elamma,pmello,ptorroni}@deis.unibo.it

Abstract. We present an architecture composed of several cooperating intelligent logic agents, *ALIAS*. In *ALIAS*, agents are equipped with hypothetical reasoning capabilities, obtained by means of abduction. A distributed abduction algorithm (*DAA*) is the mechanism agents are provided with in order to ensure consistency among the assumptions they make while trying to solve their goals (i.e., to give explanations to observations). To this purpose, a set of integrity constraints is used - together with program clauses - to confirm or discard new hypotheses. The union of agent knowledge bases ends up to generate a *global* knowledge base, which can change in a dynamic fashion, as a consequence of agent movements. In fact, although agents may keep their own *local* knowledge, they can join bunches where all the participants must agree on a set of hypotheses currently assumed. This work mainly focuses on the role played by *rambling agents* (i.e., agents moving among bunches) in composing the global knowledge of a bunch; on this subject an example is given. Several open issues are also presented.

1 Introduction

In this work, we present Abductive LogIc Agents System (from now on, *ALIAS*), an architecture composed of several cooperating intelligent agents, and the main topics currently investigated. In *ALIAS*, agents are equipped with hypothetical reasoning capabilities, obtained by means of abduction. The union of their knowledge bases ends up to generate a *global* knowledge base, which can change in a dynamic fashion, as a consequence of agent movements. In this framework, agents can perform standard deduction and also *abduce* new hypotheses, provided that they are consistent with the knowledge of other agents, i.e., the global knowledge represented by the union of the agent logic programs is maintained coherent as it would be if it was owned by a single entity. To this purpose, a mechanism to coordinate agent reasoning is introduced. Agents in *ALIAS* are grouped into bunches; each bunch represents an *agora* where agents can discuss about common arguments and/or cooperate for solving particular problems. *ALIAS* agents can move from bunch to bunch at runtime (for this reason we call them *rambling* agents), and collect information (the agent's *experience*) derived from the interactions with other agents in different agoras. More in detail, each agent is characterized by statically defined local knowledge represented by an abductive logic program [12] and possibly by a dynamically built set of assumptions (i.e., its experience). Each bunch is represented by a set of agents and a global knowledge (i.e., the set of hypotheses assumed so far within the bunch) that is dynamically built. While static knowledge is peculiar to each agent and might differ from agent to agent, all agents in the same bunch must agree on the global set of assumed abducibles. To this purpose, a set of integrity constraints is used - together with program clauses - to confirm or discard new hypotheses. Each rambling agent can improve its experience by rambling from bunch to bunch and collecting the set of hypotheses produced in the visited bunches so far. Therefore structuring agents in bunches can be seen as a way to dynamically composing several abductive logic programs. In that sense, we can consider *ALIAS* as a set of components (the bunches) whose configuration can change at runtime. Each agent entering a bunch enlarges its scope by acquiring the capability of arguing with a new set of agents.

Abduction is well known to be a suitable mechanism to detect inconsistencies in presence of incomplete, multiple and/or conflicting information. Extending an abductive algorithm to a multi-agent environment, as done in *ALIAS*, could be useful to cope with complex problem solving in distributed systems. The paper is organized as follows: Section 2 presents the architecture of the system and introduces the idea of rambling agents. Sections 3 and 4 talk about the abduction process, both classical and distributed (the DAA), and describes the mechanisms that support agent rambling. In Section 5 we give an example of how the idea of rambling through agent bunches and the abduction mechanism itself can be exploited in an application. Section 6 talks briefly about the system based on *ALIAS* which has been implemented, Sections 7 and 8 give some references to the related work and present the work in progress on *ALIAS*.

2 The Architecture of *ALIAS*

In this section, we describe the architecture of *ALIAS*, whose scheme is presented in Figure 1. In *ALIAS*, the computation is carried on by several parallel agents that cooperate for solving goals. Each agent has its own (possibly incomplete) knowledge base, and uses abduction as a way of hypothetical reasoning. Agents are grouped into *bunches*: each bunch of *arguing* agents is associated with a set of *abducible* hypotheses (i.e., the set of hypotheses that could be possibly raised) which consists of ground literals only. Moreover, agents of the same bunch refer to the common set of hypotheses, Δ, assumed so far (the *dynamic* knowledge of the bunch). When an agent A tries to raise a new hypothesis h, the agents belonging to the same bunch cooperate with A (by means of the coordination protocol which will be explained in Section 3.1) in order to check whether h is consistent with the knowledge of the other agents and with the assumptions made so far: if all agents agree on the assumption of h, h is assumed; otherwise it is discarded.

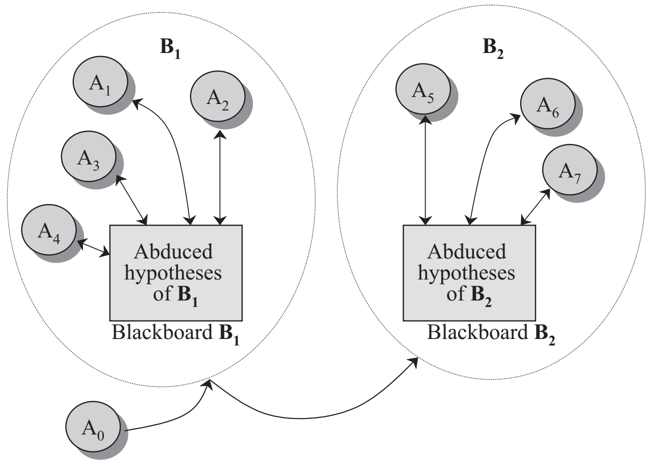

Fig. 1. The architecture of *ALIAS*

The composition of a bunch can be dynamically determined. To this purpose, *ALIAS* agents are classified in two classes: *rambling* agents and *still* agents. Each still agent is statically anchored to its bunch, and it cannot migrate to a different bunch. Its knowledge base contributes to computations occurring inside its bunch during the entire lifetime of the system. Each still agent refers to the dynamic knowledge of the bunch it belongs to, as the set of current hypotheses. Rambling agents, instead, can move from one bunch to another at runtime; moreover, each rambling agent handles a local dynamic knowledge that represents the set of hypotheses acquired during its life. For this reason, we say that rambling agents are *agents with experience.*

A rambling agent A may dynamically enter a bunch B for a variety of reasons. For instance: it might want to communicate its own experience to the bunch, thus enriching the dynamic knowledge of B; it might want to contribute to the reasoning activities of B, offering its knowledge base to the bunch; it might want to execute a query within that bunch; it might simply want to acquire the hypotheses assumed so far inside the bunch. Movements of rambling agents could be determined directly by the user, or they could be driven by higher level (e.g., planning) policies. In order to correctly deal with the dynamic management of bunches, each bunch is equipped with proper mechanisms that support both the entering and the exiting of rambling agents. In particular, the entering mechanism must guarantee the consistency of the hypotheses in the dynamic knowledge of the bunch with the knowledge base of the new agent. Some detail about the dynamic management of bunches will be given in Section 4.

3 Distributed Abductive Logic Programming

3.1 Preliminaries

Abduction plays an important role in everyday human problem solving, and it has been recognized as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge.

Given a theory T, integrity constraints I, and a possible conclusion (observation) O, an abductive explanation of O is a set of sentences Δ which together with T entails O, that is $T \cup \Delta \models O$, and such that $T \cup \Delta$ is consistent with I. Operationally, O is derivable in T with "conditional answer" Δ.

An *abductive logic program* is a triple $\langle P, \mathcal{A}, IC \rangle$ where P is a logic program possibly with abducible atoms in clause bodies; $\mathcal{A}$ is a set of *abducible predicates*, i.e., *open* predicates which can be used to form explaining sentences; IC is a set of integrity constraints: each constraint is a denial containing at least one abducible[1]. Given an abductive program $\langle P, \mathcal{A}, IC \rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of atoms Δ which together with P entails G. It is also required that the program $P \cup \Delta$ is consistent with respect to IC.

Example 1. Let us consider the following program, inspired by [19]:

```
grass_is_wet  :- rained_last_night.
grass_is_wet  :- sprinkler_was_on.
shoes_are_wet :- grass_is_wet.
```

with integrity constraints:

```
:-  rained_last_night, grass_is_dry.
:-  sprinkler_was_on, grass_is_dry.
```

Let predicates `rained_last_night` and `sprinkler_was_on` be abducible. The observation `shoes_are_wet` is explained by two (minimal) sets of sentences, respectively:

$\mathtt{Abd}_1 = \{\mathtt{rained\_last\_night}\}$ and
$\mathtt{Abd}_2 = \{\mathtt{sprinkler\_was\_on}\}$

According to [10], negation as default, possibly occurring in clause bodies, can be recovered into abduction by considering the *positive version* of a program obtained by replacing negated literals of the form *not a* with a new positive atom not_a, and by adding the integrity constraint $\leftarrow a, not\_a$ to IC.

The natural syntactic correspondence between a standard atom and its negation by default is given by the following notion of complement

$$\bar{l} = \begin{cases} \alpha & \text{if } l = not\_\alpha \\ not\_\alpha & \text{otherwise} \end{cases}$$

[1] In the following, for the sake of simplicity, we consider only ground programs, thus assuming that P and IC have already been instantiated.

where α is an atom.

From now on, we will not distinguish anymore between a program and its positive version. Thus, default literals of the kind *not* p are to be considered as abducibles.

As described in Section 2, an *ALIAS* multi-agent application is composed of several agents, possibly grouped into bunches. Each bunch consists of $n+1$ agents *arguing* on a common set of abducibles. Each bunch is also associated with a dynamic knowledge, represented by the set Δ, containing the hypotheses assumed by those agents which have been dealing with the group so far. Each agent A_i, for $i = 0, \ldots, n$, encapsulates an abductive logic program $\langle P_i, \mathcal{A}_i, IC_i \rangle$. We suppose that each integrity constraint in IC_i of an agent A_i has at least one abducible in the body. We suppose that abducible predicates have no definition as in [14]. As concerns integrity constraints, the user-defined ones are partitioned among the various agents, while those for handling negation as default, like the constraint $\leftarrow p, not\ p$, are left implicit and replicated in each agent's knowledge base. The set Δ contains the hypotheses so far assumed by all agents.

In *ALIAS*, agent reasoning and coordination are integrated within the Distributed Abduction Algorithm (DAA, for short) which extends the approach presented in [14] towards distribution.

The *DAA* algorithm we adopt for abductive reasoning in a multi-agent system is based on a proof procedure, defined originally in [10] by Eshgi and Kowalski and further refined by Kakas and Mancarella [14], which is correct with respect to the abductive semantics defined in [4], and which is described in detail in [6]. The proof procedure presented in [14] extends the basic resolution mechanism adopted in logic programming by introducing the notion of *abductive* and *consistency* derivation. Intuitively, an *abductive* derivation is the usual logic programming derivation suitably extended in order to consider abducibles. When an abducible atom h is encountered during this derivation, it is assumed, provided this is consistent. The consistency check of a hypothesis, then, starts the second kind of derivation. The *consistency* derivation verifies that, when the hypothesis h is assumed and added to the current set of hypotheses, any integrity constraint containing h fails (i.e., the bodies of all the integrity constraints containing h are false).

During this latter procedure, when an abducible L is encountered, in order to prove its failure, an abductive derivation for its complement, $\overline{L}$, is attempted. The *DAA* algorithm extends the Kakas and Mancarella approach in the sense of distribution: now knowledge is distributed among various agents, and consistency derivations have to be coordinated within a pool of logic agents. In that sense, we say that the whole Logic Program (LP) embedded in a bunch architecture is not just the union of the agent LPs embedded in a single agent. In fact, since the consistency derivations run locally, the Prolog clauses of an agent LP are not globally visible, and are not accessed in the way they would be if they were part of one single LP. On the other hand, separation among the agent knowledge bases does not involve the set of currently abduced hypotheses, which are globally visible, nor the integrity constraints (ICs), which operate globally. We say that

ALIAS implements a *union* of logic programs, but just as far as the hypotheses assumed to be true and the ICs are concerned.

In addition, rambling agents may have *experience*: this means that each rambling agent A_i encapsulates the set of hypotheses δ_i collected so far, as the result of having visited one or more bunches. Of course, by a more general point of view, an agent A_i could be processed by a higher level layer as the agent is out of any arguing bunch, e.g. in order to follow data management policies based on persistence, relevance or other attributes of the assumed hypotheses. It is worth noticing that for each bunch, the set Δ has the following property: $\bigcup_i \delta_i \subseteq \Delta$, where δ_i is the experience of agent A_i of the bunch. The set of program clauses and integrity constraints might differ from agent to agent, but we assume that the set of abducible predicates (default predicates included) is the same for all the agents in a bunch. This last hypothesis is not restrictive, since - by a syntactical point of view - it is a matter of plain implementation to set up a mechanism for communicating an agent set of abducibles together with the other information he gives while it enters a bunch. At a higher level we could also give it a semantics and imagine a naming service for the abducibles, but it does not strictly concerns our subject.

Let us give a short example in order to show how the DAA works.

Example 2. Let us consider a system composed of four agents (A_0, A_1, A_2, A_3) whose knowledge base is structured as follows:

$$A0)\ \mathtt{s} :- \mathtt{p}. \qquad A1)\ :- \mathtt{q}, \mathtt{not\ b}. \qquad A2)\ :- \mathtt{p}, \mathtt{not\ c}. \qquad A3)\ \begin{array}{l}\mathtt{r} :- \mathtt{q}.\\ \quad :- \mathtt{b}, \mathtt{c}.\end{array}$$

where p, q, b, and c are abducible atoms. Let the current set of hypotheses Δ be empty. Suppose that agent A_0 raises the goal $\leftarrow \mathtt{s}$ and agent A_3 the goal $\leftarrow \mathtt{r}$. Each single request is processed in parallel, producing two different and conflicting sets of hypotheses ($\{\mathtt{p}, \mathtt{c}, \mathtt{not\ b}, \mathtt{not\ q}\}$ for A_0 and $\{\mathtt{q}, \mathtt{b}, \mathtt{not\ c}, \mathtt{not\ p}\}$ for A_3).

4 Dynamic Management of Bunches

Openness, in general, is a welcome property in any architecture under development, and for the sake of openness, our aim is to allow agent bunches to modify dynamically their composition, by letting agents enter/exit bunches at runtime. A dynamic management of bunches requires the definition of a protocol, whose description represents the main issue of this section.

4.1 Entering a Bunch

A major feature of *ALIAS* is to allow an agent logged in a bunch to keep its own coherent knowledge representation, which could possibly be in contrast with those of its bunch partners. The only point in which the agents of a bunch must agree is that they are referring to a global dynamic knowledge, that is, the dynamic set of abduced hypotheses, Δ. The Distributed Abduction Algorithm guarantees that Δ is always consistent with each agent's integrity constraints.

A problem rises when an agent, A_0, which did not participate in formulating Δ, wants to join the bunch. In fact, A_0 might contain some integrity constraints which are in contrast with Δ itself. The dual problem rises when an incoming agent has got its own set of abduced hypotheses, δ_0, and wants to join the bunch proposing it to accept δ_0. In that case, δ_0 has to be the object of a distributed consistency check step.

By a general point of view, agent knowledge is constituted by an abductive logic program plus a set of assumed hypotheses δ, representing the history of the agent itself. Given an incoming agent A_0, let us call $\mathcal{KB}_0 = \langle P_0, \mathcal{A}_0, IC_0, \delta_0 \rangle$ its knowledge. Let $\mathcal{A}$ be the set of abducibles of the bunch and Δ the current set of abduced hypotheses within the bunch. For the sake of simplicity, we assume $\mathcal{A}_0 \equiv \mathcal{A}$. The protocol A_0 shall follow in order to join the bunch is composed by two phases: the first phase take place within A_0, while the second one involves all agents in the bunch, including A_0. Preliminarily, A_0 asks the bunch for being logged in. Then, the bunch waits until all the running computations terminate, prevents other distributed abductions from being started during agent entering and returns the current Δ to A_0. The first phase of the protocol checks if Δ is consistent with δ_0[2]. This requires the *local* evaluation of the query $\Delta \cup \delta_0$, obtained by a local abduction process within A_0. If the query fails, A_0 is not let in the bunch. Otherwise, a new additional set of hypotheses Δ_1 is obtained, such that $P_0 \cup IC_0 \cup \Delta_1 \cup \delta_0 \models \Delta$. Afterwards there starts a second phase of the protocol, whose aim is to check the consistency of $\delta_0 \cup \Delta_1$ in the whole bunch. If $\delta_0 \cup \Delta_1$ is empty, there is nothing to discuss among the agents of the bunch, and A_0 is simply let in. Otherwise a distributed abduction process is started by A_0 itself in order to find a Δ_2 which together with the global knowledge entails $\delta_0 \cup \Delta_1$. Again, this step could succeed or not: in this latter case A_0 will not enter the bunch. Otherwise, A_0 is confirmed part of the bunch, and the global knowledge becomes $\Delta \cup \delta_0 \cup \Delta_1 \cup \Delta_2$.

In plain words, we could say that Δ_1 represents the hypotheses A_0 must add to his δ_0 if he wants to get in, and that Δ_2 is the result of the negotiation of the whole set of hypotheses once A_0 is let in the bunch. The protocol is in that sense two-fold (the first phase joining the agent δ_0 with the bunch Δ and the second one completing them with additional consistency information).

4.2 Exiting a Bunch

There could be two different kinds of exiting of one agent from a bunch: a *traceful* leaving and a *traceless* leaving.

A traceful leaving implies that an agent A_0 exiting a bunch B leaves in the set Δ associated with B a set of hypotheses abduced with its contribution, i.e.,

[2] Given two sets of abduced hypotheses, Δ_a and Δ_b, they are considered *syntactically* consistent if their union $\Delta_a \cup \Delta_b$ does not contain both a literal and its negation.

the hypotheses are also supported by A_0 knowledge base. In that case, A_0 has only to notify the event to the bunch and then it can exit.

In the case of a traceless leaving, an agent A_0 going out of B doesn't leave any hypothesis assumed with its contribution within the bunch after its exiting. In this case, the exiting protocol is more complicated, since the introduction of some form of belief revision is required. The definition of a proper protocol for the traceless exiting is subject for current and future work.

As A_0 exited B, a process of filtering could run on the *experience* matured by the agent during its staying in the bunch. In general, we could think to associate some attributes with abducibles in order to provide them with a meta-level semantics to be of support to a higher level reasoning (e.g., *planning*) module.

5 An Example: Navigating Through the *Abductive Web Travel Agency*

This section aims at showing a possible use of the architecture we have presented. Anybody who deals with the Internet knows how spontaneous it is nowadays to navigate around in order to find some place to spend the holidays. Thus, we considered, as a possible application of *ALIAS*, an environment composed of several bunches, being part of a distributed architecture which implements a *Web Travel Agency.*

When a customer enters a traditional Travel Agency she/he is usually asked which are her/his desiderata and needs, the maximum expense and so on (we could call it an *input*), and is offered as an *output* a more or less wide range of possible travels. Most of the times there is no point in asking the customer a detailed description of the input, for the employee could try and imagine the customer's preferences, or could also consider more than one possible profiles building up different suitable scenarios. For example, a profile could be that of a customer who likes the mountains and wants to sleep in a comfortable hotel room, another one prefers some adventurous off-limits holidays and does not care sleeping in a tent while is ready to spend a fortune to see the lions, a third one has got children and just wants to have some rest, and of course appreciates very much any kind of baby-sitting facilities. The perfect travel agency employee has to cope with last minute offers and deals made in the background and does a just-in-time planning for a wide range of possible customers. Which is obviously quite a hard task, requiring good coordination skills and a lot of fantasy. That scenario is also difficult to scale, for everything is managed by a single person.

Now, let us consider a Web customer sitting in front of a screen, who bumped in the site of the *Abductive Web Travel Agency* ^{TM}. He/she is asked to fill in a form, giving a rough sketch of its desiderata and needs, which he/she could either specify in detail or leave blank. After a while, the Web customer is given a wide range of opportunities to choose among. They are in the form: "*If You like the adventure and the mountains and You like flying by plane, You can go and do this and that on the Rocky Mountains - click here to find more details - or if You want to spend less than x money and you are interested in Archeological stuff and* *you do not care moving by bus You can go and discover the ancient civilizations of Yucatan on the second half of February ...*", and so on. In other words, the system made certain hypotheses on the customer's place, and assumed to be true something he/she did not either think about. How many different scenarios can be created and explored that way, with no need of a priori knowledge from any employee at all! How does it work?

Under the Abductive Web Travel Agency form, which is accessible via Web, lies a syntactical translator which transforms the information given by the user into a Logic Program. For example, let us consider a family composed by two adults and two kids of six and three, who wants to relax on the beach from the 7^{th} to the 14^{th} of August who does not want to spend too much but does not want to go camping either. For such a customer, a logic program could be the following list of assumptions:

```
under_4.
class_economy.
not camping.
beach.
high_season.
not business.
```

Those assumptions constitute the δ of an agent that will be created for the customer. The agent could also be provided with a more complex logic program and integrity constraints, specifying some policies or needs of the Web Travel Agency. Possibly, the web site of the travel agency has also a certain number of integrity constraints and a logic program, to which respect the agent δ becomes the query to solve. For instance,

```
:- under_4, off_limits.
:- over_60, off_limits.
:- class_business, camping.
:- class_business, bus.
:- relax, bus.
:- class_economy, plane, high_season.
:- class_economy, class_business.
:- under_4, not baby_sitting_service.
:- off_limits, relax.
```

with obvious semantics. As a consequence of running the query δ on that logic program, we obtain a new δ:

{`under_4,class_economy,not camping,beach,high_season,not business,not class_business,not off_limits,not plane,baby_sitting_service`}.

At this point, the Web Travel Agency clones n times the agent it created for the customer, and sends those Rambling Agents to some different bunches, each of them managing a certain kind of travel: for instance, the "Adventure Bunch", the "Business Bunch", the "Art Bunch", the "Relax Bunch", and so on.

An agent submits its δ as a query to the bunch it wants to access, and can be either rejected or let in. In our example, we may suppose that the agents are rejected from the "Adventure Bunch" and from the "Business Bunch". Entering a bunch causes an enlargement of the agent δ, for instance after entering the "Relax Bunch" the hypothesis {`relax`} is added. Obviously, we assume that the agents do not modify the bunch Δ (that is, a policy of *traceless* leaving is adopted). After one or more of those steps, which guide the agents through a multitude of bunches, those agents which still live get to some places where a certain number of destinations is proposed, for instance we can imagine that a rambling agent who left the "Relax Bunch" is lead to another bunch which handles the following knowledge:

```
suitable_plan :- saint_tropez.
suitable_plan :- taormina.
suitable_plan :- rimini.
suitable_plan :- marrakech.
suitable_plan :- dolomiti.
saint_tropez :- beach, class_business.
taormina :- beach, history_and_art.
rimini :- beach.
marrakech :- folklore, plane.
dolomiti :- mountains.
```

and tries to solve the query `:- suitable_plan.` as soon as the rambling agent is let in. The agents which are finished get back to the Web Travel Agency Home Page where they show their acquired knowledge in order for the Web Travel Agency to produce a list of readable, user-friendly choices.

We could imagine a lot of examples like that, since agent based programming often refers to some *real world*-like behaviour, and the hypothetical reasoning which lead us to the adoption of an abduction mechanism is just what the human brain constantly performs in the background. Related fields of application could be the discussion groups, or any environment in which an agent might be let in or not depending on some characteristics it has, or it misses, if we represent them by means of a set of abducibles which the agent takes with him as a "signature", that is, its δ.

6 The Implementation

The implementation of *ALIAS* has been obtained by using Java and *Amzi!* Prolog [2]. Agent can be created, given a logic program, and guided through the blackboards through a User Interface, which is also used to trigger the Distributed Abduction Algorithm by asking a query to an agent. Java is used to implement the framework of the system, and in particular for supporting the communication between agents, while Prolog is used to implement both abductive and consistency derivation local mechanisms. Details on the implementation are given in [6]; a Home Page of *ALIAS* which is set up for downloading the software is also kept up to date [1].

7 Related Work

In this section, we first survey some related work concerning both sequential approaches to abductive reasoning and then mention related (distributed) multi-agent systems based on the logic programming paradigm.

The topic of modular logical programs is discussed in [5], where a semantics for the composition of logic knowledge bases is proposed. *ALIAS* implements to some extent a (dynamic) composition of integrity constraints and abducibles, while leaves the knowledge bases of the agents composing the bunch separated: in fact there is no global visibility of the agent logic programs, since consistency checks are performed locally.

The abductive coordination protocol introduced in Section 3.1 is grounded on a query-based proof procedure originally defined by Eshgi and Kowalski [10] that recovers negation as default through abduction, and further extended by Kakas and Mancarella in [14] in order to manipulate arbitrary abducibles and integrity constraints. In [4] this proof procedure has been further extended to consider a second kind of negation, and proved sound and weakly complete with respect to a three-valued abductive semantics.

Other relevant literature exists on proof procedure for extended and abductive logic programs. Satoh and Iwayama [23] proposed a query-based proof procedure which improves Kakas and Mancarella's one, and addresses completeness with generalized stable model semantics [13], but at the expense of a more complex integrity checking. Another proof procedure has been proposed by Denecker and De Schreye in [8], by extending the SLDNF resolution to the case of abduction. A crucial property of this abductive procedure is the treatment of *non-ground* abductive goals considered neither in [14] nor in our abductive protocol. In [8], the authors do not consider general integrity constraints, but only constraints of the kind $\leftarrow a, not\ a$. To overcome this limitation, in a later work [9], they consider the treatment of general integrity constraints that are checked only at the end of the proof for a query, i.e., only when the overall set of abductive hypotheses supporting the query has been computed. Nonetheless, it is better to check consistency as soon as the hypotheses are generated.

The problem of incremental consistency checking for the generated hypotheses (possibly containing variables) is considered in *Theorist*, by Poole [20]. In *Theorist*, consistency is immediately checked for those hypotheses with no variables and delayed until the end for hypotheses containing variables at the time they are generated.

A parallel implementation of the proof procedure described in [14] has been proposed by Kakas and Papadopoulos in [15]. Various forms of parallelism (*OR* and dependent *AND* parallelism) are exploited to parallelise deductive logic programming in the framework of abductive logic programming. This parallel model for abduction is then mapped into the Andorra model. Nonetheless, the focus of [15] is on a parallel implementation of a single abductive logic programming computation rather than on the multi-agent case.

A notion of abduction for the family of Concurrent Constraint (`cc`) languages was proposed by Codognet and Saraswat in [7]. The key idea of the `cc` framework is to use constraints (and a global set of constraints called the constraint *store*) to synchronize concurrent logic computations. Multiple agents run concurrently and synchronize by adding a new constraint to the *store* (*tell* operation) or checking whether a given constraint is entailed by the *store* (*ask* operation). An asking agent may block and suspend in information in the *store* is insufficient to determine whether the constraint is entailed or not. In (`cc`) computations this corresponds to a situation where only partial information in the *store* is available. In [7], thus abduction is used in order to generate some hypothetical constraint that would lead some *ask* operations to be satisfied and therefore resume some agent computations. With respect to our work, they consider constraints instead of atoms as hypotheses, and the consistency ckeck is delayed, during the computation, till some other agents will add to the store a possibly inconsistent constraint.

Relevant work concerning logic agents is contained in [16, 17], where Kowalski and Sadri analyse differences between rational and reactive architectures. Rational agents have both beliefs and goals. A reactive agent, instead, has neither an explicit knowledge base, nor an explicit representation of goals. In [17], the authors propose a uniform agent architecture that aims to capture both rationality and reactivity. In a rational agent, goals are represented explicitly and knowledge is represented as goal reduction rules of kind *conclusion if condition*. Furthermore, condition-action rules, peculiar of reactive agents, can be interpreted as integrity constraints. The two kinds of behaviour are then unified in a single logic-based architecture which is grounded on a proof procedure where integrity constraints and queries are treated identically [18]. This proof procedure is general, and has been shown to unify abductive logic programming, constraint logic programming and semantic query optimisation.

According to the definitions above, our logic agents are to be considered rational agents, since their main capability is to answer queries (i.e., goals) from the environment (i.e., other arguing agents) or high-level goals. Rather than on agent's behaviour, the emphasis in our work is on the definition of an abductive coordination protocol inside a multi-agent system. In this respect, our work can be considered complementary to [16, 17].

As concerns dynamicity, and dynamic creation of agents in particular, a relevant work is represented by *Linear Objects* [3]. In *Linear Objects*, dynamic agent creation is supported via a built-in primitive which clones the tuple space. The integration of such a feature in our system would greatly improve the system and is subject for future work.

Finally, rambling agents introduce in *ALIAS* the issue of *mobile agents*, which is widely researched in both academy and industry. The principles underlying the idea of code mobility are explained in [11]; some more specific papers could be found in [22, 21]. To the best of our knowledge, no work on abductive mobile agents has been published so far.

8 Conclusion and Future Work

We have presented *ALIAS*, an architecture composed of several cooperating abductive logic agents. We defined a basic protocol to coordinate the reasoning of abductive (rational) agents, in order to introduce abduction in a (logic) multi-agent environment. We focused on the issue of dynamically composing the bunches of arguing agents, presenting a mechanism aimed at guaranteeing coherence in the global knowledge of a bunch after a new agent is let in. The architecture and the mechanisms described have been implemented and tested in order to verify their correctness and to test the efficiency of the system [6].

The introduction of rambling agents proposes a multitude of open issues, in particular those concerning persistence of abduced hypotheses and the monotonicity of the abduction process. An interesting work could be to study a classification of rambling agents by a semantic point of view focusing on the possible role of agents in the bunches. The results of such classification work may be helpful to application designers while mapping user specifications into *ALIAS* bunches and agents. Since rambling agents are the caretakers of abductive logic programs, that may lead to interesting results in the field of knowledge representation, as we could interpret bunches as *loci* in which a certain knowledge representation is kept consistent and coherently updated. Different theories could be generated in different bunches by putting together here and there some agents rather than other ones.

In the future, a main research issue will concern the way of extending the Distributed Abduction Algorithm in order to allow non-ground literals to be part of the set of abducibles, thus extending the system and providing a likely support for real applications. We also intend to study the integration of the abductive reasoning module with a higher level planning module in order to define a suitable architecture for complex high level problem solving.

9 Acknowledgements

This work has been supported by M.U.R.S.T. Project on *Intelligent agents: interaction and knowledge acquisition*.

References

1. *Abductive LogIc Agent System.* http://www.lia.deis.unibo.it/Software/ALIAS/.
2. AMZI! *Amzi!* Prolog + Logic Server 4.1. http://www.amzi.com.
3. J-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. In *Proc. ICLP90*, MIT Press, 1990.

4. A. Brogi, E. Lamma, P. Mancarella and P. Mello. A Unifying View for Logic Programming with Non-Monotonic Reasoning. In *Theoretical Computer Science*, Vol. 184, pages 1-49, North Holland, 1997.
5. M. Bugliesi, E. Lamma and P. Mello. Modularity in Logic Programming In *The Journal of Logic Programming*, Vol. 19/20, pages 443-502, North-Holland, 1995.
6. A. Ciampolini, E. Lamma, P. Mello, C. Stefanelli and P. Torroni. An Implementation for Abductive LogIc Agents. In *Proc. AI*IA99*, Pitagora Editore, 1999 (to appear).
7. P. Codognet and V. Saraswat. Abduction in Concurrent Constraint Programming. INRIA-Roquencourt Tech. Rep., March 1992.
8. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In K. Apt, editor, *Proc. Int. Joint Conf. and Symp. on Logic Programming*, pages 686–700. The MIT Press, 1992.
9. M Denecker and D. De Schreye. Representing Incomplete Knowledge in Abductive Logic Programming. In *Proc. Int. Logic Programming Symp. ILPS93*, pages 147–163. The MIT Press, 1993.
10. K. Eshgi and R.A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. Sixth Int. Conf. on Logic Programming*, page 234. The MIT Press, 1989.
11. A. Fuggetta, G.P. Picco and G. Vigna. Understanding Code Mobility.*IEEE Trans. on Software Engineering*, 24(5), May 1998.
12. A.C. Kakas, R.A. Kowalski and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
13. A.C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. of 9th European Conf. on Artificial Intelligence.* Pitman Pub., 1990.
14. A.C. Kakas and P. Mancarella. On the relation between Truth Maintenance and Abduction. In *Proc. PRICAI90*, 1990.
15. A.C. Kakas, and G.A. Papadopoulos. Parallel Abduction in Logic Programming. In *Proc.Int. Symp. on Parallel Symbolic Computation PASCO94.* pages 214-224, World Scientific Pub., 1994.
16. R.A. Kowalski. Using metalogic to reconcile reactive with rational agents. In *Meta-Logics and Logic Programming*, K. Apt and F. Turini (eds.), MIT Press, 1995.
17. R.A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. Int. Workshop on Logic in Databases*, San Miniato, Italy, Springer-Verlag, LNCS 1154, 1996.
18. R.A. Kowalski, and F. Toni and G. Wetzel. Towards a declarative and efficient glass-box clp language. In *Proc. of Logic Programming Workshop WLP94*, N. Fuchs and G. Gottlob (eds.), 1994.
19. J. Pearl. Embracing causality in formal reasoning. In *Proc. National Conf. on Artificial Intelligence*, Seattle, WA, pages 369-373, 1987.
20. D. Poole. Compiling a Default Reasoning System into Prolog. In *New Generation Computing*, 9(1)3–38. Springer-Verlag 1991.
21. *Proc. Joint Symp. ASA/MA'99 of 1st Int. Symp. on Agent Systems and Applications (ASA'99) and 3rd Int. Symp. on Mobile Agents (MA'99).* Palm Springs (CA), USA, October 3-6, 1999. IEEE Press (to appear).
22. K. Rothermel and F. Hohl (eds.). *Proc. 2nd Int. Workshop on Mobile Agents.* Lecture Notes in Computer Science. Springer-Verlag, Vol. 1477, September 1998.

23. K. Satoh and N. Iwayama. A Query Evaluation Method for Abductive Logic Programming. In K. Apt, editor, *Proc. Int. Joint Conf. and Symp. on Logic Programming*, pages 671–685. The MIT Press, 1992.